Never stop thinking

TriCore

AP32152

Concurrent multi-threaded execution

Application Note

V1.0 2010-03

Microcontrollers

Edition 2010-03

Published by
Infineon Technologies AG
81726 Munich, Germany

LEGAL DISCLAIMER

THE INFORMATION GIVEN IN THIS APPLICATION NOTE IS GIVEN AS A HINT FOR THE IMPLEMENTATION OF THE INFINEON TECHNOLOGIES COMPONENT ONLY AND SHALL NOT BE REGARDED AS ANY DESCRIPTION OR WARRANTY OF A CERTAIN FUNCTIONALITY, CONDITION OR QUALITY OF THE INFINEON TECHNOLOGIES COMPONENT. THE RECIPIENT OF THIS APPLICATION NOTE MUST VERIFY ANY FUNCTION DESCRIBED HEREIN IN THE REAL APPLICATION. INFINEON TECHNOLOGIES HEREBY DISCLAIMS ANY AND ALL WARRANTIES AND LIABILITIES OF ANY KIND (INCLUDING WITHOUT LIMITATION WARRANTIES OF NON-INFRINGEMENT OF INTELLECTUAL PROPERTY RIGHTS OF ANY THIRD PARTY) WITH RESPECT TO ANY AND ALL INFORMATION GIVEN IN THIS APPLICATION NOTE.

Information

For further information on technology, delivery terms and conditions and prices, please contact the nearest Infineon Technologies Office (**www.infineon.com**).

Warnings

Due to technical requirements, components may contain dangerous substances. For information on the types in question, please contact the nearest Infineon Technologies Office.

Infineon Technologies components may be used in life-support devices or systems only with the express written approval of Infineon Technologies, if a failure of such components can reasonably be expected to cause the failure of that life-support device or system or to affect the safety or effectiveness of that device or system. Life support devices or systems are intended to be implanted in the human body or to support and/or maintain and sustain and/or protect human life. If they fail, it is reasonable to assume that the health of the user or other persons may be endangered.

TC1797

Revision History: V1.0, 2010-03

Previous Version: none

Page	Subjects (major changes since last revision)

We Listen to Your Comments

Is there any information in this document that you feel is wrong, unclear or missing?
Your feedback will help us to continuously improve the quality of this document.
Please send your proposal (including a reference to this document) to:

mcdocu.comments@infineon.com

Table of Contents

1 Preface

This application note describes a basic thread-support implementation on the TriCore architecture [1]. The document is aimed at developers who write or design real-time applications on the TriCore and are considering splitting the application into multiple threads of execution. This document looks specifically at those features of the TriCore architecture that make it such an attractive platform for real-time embedded systems, focusing particularly on fast context switching and the use of software-posted interrupt requests for low-overhead interfacing to the task-dispatch function of the kernel.

This guide assumes that readers have access to the TriCore Architecture Manual [1], and have at least some general knowledge of the TriCore instruction set and architectural features. Those chapters of the Architecture Manual covering Core Registers, Context Management and function-calling are particularly pertinent to potential readers of this document.

See References on Page 25 for more information on the TriCore Architecture manual and other relevant documentation.

It is assumed that most readers will be generally familiar with the features and functions of Real-Time Operating Systems (RTOSs). However, between this document and those listed in the references, there is enough information to enable embedded systems programmers with no prior RTOS experience to understand and add custom RTOS functionality to embedded applications that require it.

2 Introduction

This chapter explores specific real-time issues for the TriCore architecture and explains how particular features of the architecture can be used to expedite task management and scheduling.

Processing is performed either at the interrupt service level, with context switching driven entirely by the priority-interrupt mechanisms of the hardware, or at a non-interrupt level by software-managed tasks. The interrupt system still operates of course, but in addition to the context switching involved in taking and returning from an interrupt, there is software-managed context switching between application tasks, driven by the task management of a real-time kernel.

Typical deeply embedded applications on the TriCore are driven mainly by interrupts. The TriCore architecture supports these systems with multiple features, resulting in extremely low interrupt latency.

- Hardware-supported context switch. The upper context, sixteen 32bit registers (512bits) are stored in 2 CPU cycles. While application programmers are limited to 64-bit load/store operations[1], the core stores 128 bits in one cycle. During a context load/store, a shadow bank inside the TriCore is used to temporarily store 256 bits, so that although the complete store operation will take 4 cycles, the core is ready to execute the interrupt routine within 2 cycles.
- 4-stage pipeline. The TriCore has a 4-stage, fetch/decode/execute/writeback pipeline. When an interrupt is taken, the first instruction is executed after just 3 cycles.
- The Interrupt Control Unit (ICU) is running independently from the core. The core execution is broken only when an interrupt is pending which has a higher priority than the current priority of the core. The ICU can handle up to 255 priorities. By default, interrupt routines should be short and non-interruptable. It's up to the application programmer to enable higher-priority interrupts during interrupt execution. Interrupts can be grouped.
- TriCore derivatives implement a second core on the same silicon. The Peripheral Control Processor (PCP) is directly connected to the peripheral bus, has its own ICU, and can handle interrupts independently from the TriCore or can preprocess values before sending them to the TriCore. This offloads simple tasks from the main CPU.

The TriCore interrupt system is very efficient, fast and small. Systems that are solely built on interrupts without using any software-managed tasks, are referred to as bare hardware systems. However, bare hardware systems are limited in the complexity of the applications they can handle, as the different Interrupt Service Routines (ISRs) have little scope for interaction. They may share global variables for communicating status, but are con-

[1] 128-bit load/store will be available on TC1.6

strained to a strict priority-based, last-in, first-out scheduling sequence. An ISR can not be allowed to suspend its execution to wait for a message from another ISR because it would be implicitly blocking execution of any lower-priority ISR that it had interrupted. In real-time kernel based systems, the routines that service hardware interrupts are typically small and fast. Their main function is to capture or send data, and to notify the task-scheduling kernel of any further processing required. The bulk of application processing is carried out by tasks running at the “background”’ level of the processor; i.e. its normal state when it is not executing an ISR. There are two general types of task scheduling in real-time kernel based systems:

- Non-preemptive scheduling,
- Preemptive systems scheduling.

2.1 Non-Preemptive Scheduling

In non-preemptive scheduling, the kernel never preempts a running task in order to switch to another task. Some explicit action is required by the running task for it to give up control of the CPU. For example, the task may request a resource that is not currently available, with an implied request to suspend its execution until the resource becomes available. The kernel would then remove the task from the ready queue, and look for another task to which to pass control of the CPU.

Non-preemptive scheduling is sometimes referred to as co-operative multitasking, because tasks co-operate with one another to pass control of the CPU. A running task can be interrupted, but the interrupt will not cause it to lose control of the CPU to another task. An ISR will always return control to the task or service routine that was running when the associated interrupt was taken. That makes a non-preemptive task management kernel simpler and easier to write than a preemptive kernel. More importantly it also simplifies certain aspects of the application tasks. There is much less need to guard access to shared data through the use of semaphores, because each task executes for as long as it needs, without concern for preemption. For the same reason, tasks may call non-reentrant functions that use statically-allocated, temporary variables.

A further potential advantage of non-preemptive scheduling is that interrupt latency can be lower than with pre-emptive scheduling. Non-preemption largely eliminates the need for interrupt lockouts within critical code sections. In addition, the de-coupling of interrupts from task scheduling means that interrupt servicing does not go through the kernel and ISRs avoid a level of overhead that they would otherwise incur. The overhead of en-queuing a task request and exiting through the kernel to get the task dispatched is avoided.

On the other hand however, while the ISR is executing, lower-priority interrupts are implicitly disabled. The ISR is subject to the same non-blocking considerations and consequent limitations on inter-module messaging that it would have in a bare hardware system. Therefore in practice most ISRs remain limited to data capture and posting of event notifications. However, in a non-preemptive system there is no guarantee as to when a running task will get around to checking for the notification, or make a call that will allow the kernel to do so. That means that if the overall system of co-operating tasks and ISRs is not carefully designed, “non-preemptive” can easily become “non-responsive.” For that reason non-preemptive kernels are rarely used in hard real-time systems.

2.2 Preemptive Scheduling

Virtually all commercial real-time operating systems on the market implement preemptive scheduling. That means that an interrupt can cause a waiting task to become ready to execute, and if the new task has higher priority than the task running when the interrupt was taken, the new task will be dispatched when the ISR exits, preempting the previously running task.

The simplest model for preemptive scheduling is:

1. Application task is running at non-interrupt (background) level. All interrupts are enabled.
2. Interrupt is taken: ISR calls real-time kernel function (event notification) that results in a change to the task-ready queue.
3. Real-time kernel updates the appropriate queues, but takes no further action before returning to the ISR.
4. ISR completes but does not issue a direct return from interrupt. Instead, it branches to a kernel entry point or ISR termination.
5. The kernel, still operating at the interrupt level from which it was invoked, examines the task-ready queue. If a task switch is called for, it modifies the return context for the interrupt, such that a switch is made to the new context. The kernel then executes the return from interrupt.

This model assumes that ISRs execute to completion, without being interrupted in turn by higher-priority events. In practice, keeping interrupts disabled for the duration of an ISR may be unacceptable in terms of impact on interrupt latency. So the model must deal with two complications:

- Determining when to invoke the task-dispatch function. It must be invoked just prior to returning from interrupt to user level, but not when returning from one interrupt level to another.
- Guaranteeing that when the task-dispatch function has selected the highest priority ready task, it can complete the dispatch of that task without another interrupt coming in and invalidating its selection.

Besides implementing the dispatch operation itself (i.e. a task context switch), handling these two complications is the most crucial function of any real-time executive or operating system. The code for handling them is literally the “kernel” around which the rest of the system is built.

3 Task Management on TriCore

This chapter explores specific RTOS porting issues for the TriCore architecture and explains how particular features of the architecture can be used to expedite task management and scheduling.

3.1 Task Dispatch Control

For many architectures, determining when to invoke the task-dispatch logic is non-trivial. From an ISR there is no immediate way of knowing whether the current return context is for a software-managed task, or for another ISR that was executing when the current interrupt was taken.

One way to deal with this is to require every ISR to “register” itself with the kernel immediately upon entry, while interrupts are disabled. Registration may consist of as little as atomically incrementing an interrupt-nesting counter. The ISR is also required to exit through the kernel, giving the kernel an opportunity to test and decrement the nesting counter. When the counter value prior to decrementing is 1, the kernel knows that it is about to return to the user level. At that point it branches to the dispatcher to check whether it should return to the interrupted task, or switch to a new task.

The TriCore architecture provides a more direct method of determining the level of the current return context. The user level, at which software-managed tasks are executed, is priority level 0. At all times the CPU keeps track of both the current CPU Interrupt Priority Number (CIPN), and the CIPN of the return context. Both values can be read by software, so the test for being at the first interrupt level is simply to check for a return context CIPN of zero. This is the PCPN field (Previous CPU Priority Number) of the Previous Context Information (PCXI) register.

A limitation of this approach is that it still requires ISRs to exit through the kernel, in order to perform the check[1]. However, an alternative approach is available that avoids even that modest overhead.

Tasks and ISRs with an I/O privilege level of 2 (Supervisor mode) can post interrupt requests by writing to the control registers of interrupt nodes. If interrupt priority level 1 is left unused by any hardware device, and there is an interrupt node available that is not used for any hardware interrupt, then entry # 1 in the interrupt vector can be used to invoke the task-dispatch function. If an ISR makes a kernel call that results in a change to the task-ready queue, then the kernel will post an interrupt request at priority level 1. The ISR for that IPN will be the task-dispatch function. Since priority level 1 is below any hardware interrupt priority, but above the level for software-managed tasks, that interrupt will be taken after all hardware interrupts have been serviced, but before returning to the original interrupted task.

Using a software-posted interrupt to access the task dispatcher might initially seem to create unnecessary context-switching overhead. However, the TriCore’s interrupt system is optimized to avoid unnecessary context switching. If there is a pending interrupt at a level below the current CIPN, but above the CIPN of the return context, then the hardware executes a return from the current interrupt level as a branch to the ISR for the pending interrupt. The return from the current ISR is folded with the taking of the pending interrupt, bypassing the restore and immediate re-save of the return context.

3.2 Task Dispatching

An issue that must be addressed by the task dispatcher is to ensure that a dispatch request cannot “disappear” between the time the highest-priority ready task is identified and the time the dispatch is completed; i.e. before the return from interrupt to the dispatched task is executed by the dispatcher. The scenario for a lost request is:

1. An interrupt is taken and results in a change in the task-ready queue.
2. The task dispatcher is entered and the highest priority ready task is identified.
3. After the task is identified, but before it is actually dispatched, another interrupt is taken. That interrupt results in further change to the task-ready queue, adding a task whose priority is higher than the task that was about to be dispatched.

[1] The ISR could perform the check in-line, but it would have to know where to go depending on the outcome of the check. It is cleaner and as efficient for it to branch to an exit function in the kernel.

4. The second ISR, determining that it interrupted another ISR rather than a user routine, returns to the interrupted ISR. It does not re-enter the task dispatcher, but assumes (incorrectly) that the dispatcher will be entered after the ISR to which it is returning completes.
5. On resumption following the second interrupt, the task dispatcher completes the dispatch of the task it had previously found without noticing that a new, higher-priority task has been enabled.

A simple way to avoid this problem is for the task dispatcher to disable interrupts before locating and dispatching the highest-priority task on the ready queue. Interrupts remain disabled until the task dispatch is completed. That is in fact how most RTOS implementations do address the problem. Often the time required to find and dispatch the highest-priority task on the ready queue is the longest period during which interrupts are disabled, and it determines the worst-case interrupt latency for the RTOS.

A popular way to implement the task-ready queue is with a one or two-level bit vector. If a system has 32 or fewer software-managed tasks, each one can be assigned a unique bit in a single-word bit vector. The bits are assigned according to the priority order of the tasks. The ready queue is a single word in memory. A task is marked as ready to execute by setting its bit in the ready queue.

Finding the highest-priority task ready to execute is a matter of identifying the leftmost “1” bit in the ready queue. On the TriCore this can be done in one cycle, using the CLZ (Count Leading Zeros) instruction.

It is common practice for the kernel routines that set entries in the task-ready queue to check the priority of the entry being set against that of the currently running task. This allows them to determine if a context switch will be needed. If a switch is not required then there is no reason to go to the task dispatcher. Therefore when the dispatcher is entered, it inherently knows that it will be performing a context switch. One of the first things it will do for a TriCore implementation is to save the current value of the PCXI register, containing a pointer to the current task’s saved context. It saves it in an array of PCXI values indexed by the current task’s priority/ID number. This can be thought of as a part of the Task Control Block (TCB) of the current task, where the TCB data structure is distributed over multiple arrays indexed in parallel. After saving the current context pointer it proceeds to examine the task-ready queue.

After the dispatcher disables interrupts, it uses the CLZ instruction to find the index number of the highest-priority ready task. If there are no entries in the ready queue the value returned by CLZ will be 32, which is the index for the always-enabled system idle task. It stores that index in a global variable for reference by other kernel routines, then dispatches the task by indexing into the array of saved PCXI values, moving that value into the PCXI register, issuing an RSLCX instruction to restore the new task’s lower context and executing a Return From Exception (RFE) to complete the dispatch of the new task. The restoration of the upper context of the newly dispatched task by the RFE instruction includes loading the task’s saved Program Status Word (PSW), which re-enables interrupts. The total length of the critical dispatch sequence (Listing 1), where interrupts are disabled, is then just nine instructions (including the DISABLE instruction itself):

```
1 disable                    ; START CRITICAL SECTION
2 ld.w d0,[a0]%offset(OSReadyQ)      ; Task Ready Queue (32 entries)
3 clz d0,d0                  ; Index of highest priority ready task (32 if none)
4 st.w [a0]%offset(OSCurrentTask),d0
5 addsc.a a15,a8,d0,2        ; Address of word containing context pointer
6 ld.w d0,[a15]              ; Context pointer for this task
7 mtcr CR_INDEX_PCXI,d0      ; Move context pointer to core register PCXI
8 rlcx                       ; Restore lower context
9 rfe                        ; Complete the dispatch operation
```

Listing 1 Dispatch sequence

Note: If a software-posted interrupt is used to access the task dispatcher, it is possible to avoid disabling interrupts in the dispatcher altogether. If the sequence shown above is interrupted, and as a result of the interrupt a higher-priority task becomes ready to execute, the final RFE will not cause the initial task to be dispatched. The CIPN for its return context will be zero, while a software-posted interrupt will now be pending at priority level 1. The RFE instruction will effectively operate as a branch to the ISR for priority level 1, which is the task dispatcher itself. This second pass through the dispatcher will pick up the newly readied task and dispatch it correctly.

4 Threads

POSIX Threads, or Pthreads, is a POSIX standard for threads [4]. The standard defines an API for creating and manipulating threads. Pthreads are most commonly used on Unix-like POSIX systems. Pthreads defines a set of C programming language types, functions and constants. It is implemented with a pthread.h header and a thread library. Programmers can use the library to create, manipulate and manage threads, as well as synchronize between threads using mutexes and condition variables. This application note implements the most important functions of the pthread library for deeply embedded systems. Functions that are not portable because either the parameters are different from the standard or the function does not exist in the standard, are marked with the suffix _np.

4.1 Getting Started

Listing 2 shows a first pthread example running two threads concurrently. Two thread-control blocks (th1, th2) are defined in Lines 2-3 using the macro PTHREAD_CONTROL_BLOCK. The threads are set up with priority level 1, a round-robin scheduling policy SCHED_RR and the default stack size. The main function (Lines 23-31) initializes the PLL, creates the threads (Lines 26-27), initializes the pthread scheduler (Line 29), and starts the threads (Line 30). The threads are created calling pthread_create_np with the thread control block, the thread function, and an argument that is passed to the thread function. The thread functions thread1 (Lines 5-12), thread2 (Lines 14-21) once started run forever. The endless loop contains a busy waiting delay_ms function. thread1 is delayed by 100 ms, thread2 by 200 ms. Each thread function in this example increments a local counter variable. These variables are allocated on the thread local stack. With the busy waiting function, the thread1 counter should increment twice as fast as the thread2 counter. The counter variable is printed to the simulated I/O terminal. The counter values can be recognized on the stack with the debugger watch view (Figure 1).

Example 1

```
 1 // Define thread control block: name, priority, policy, stack size
 2 PTHREAD_CONTROL_BLOCK(th1,1,SCHED_RR,PTHREAD_DEFAULT_STACK_SIZE)
 3 PTHREAD_CONTROL_BLOCK(th2,1,SCHED_RR,PTHREAD_DEFAULT_STACK_SIZE)
 4
 5 void thread1(void* arg) {
 6     uint32_t volatile counter = 0;
 7     for (;;) {
 8         counter++;
 9         printf("Thread %d counter = %d\n",(int)arg,counter);
10         delay_ms(100);
11     }
12 }
13
14 void thread2(void* arg) {
15     uint32_t volatile counter = 0;
16     for (;;) {
17         counter++;
18         printf("Thread %d counter = %d\n",(int)arg,counter);
19         delay_ms(200);
20     }
21 }
22
23 void main(void) {
24     pll_init();
25
26     pthread_create_np(th1, NULL, thread1, (void*)1);
27     pthread_create_np(th2, NULL, thread2, (void*)2);
28
29     pthread_schedrr_init_np();
30     pthread_start_np();
31 }
```

Output

```
Thread 1 counter = 1
Thread 2 counter = 1
Thread 1 counter = 2
Thread 2 counter = 2
Thread 1 counter = 3
Thread 1 counter = 4
Thread 2 counter = 3
Thread 1 counter = 5
Thread 1 counter = 6
Thread 1 counter = 7
Thread 2 counter = 4
Thread 1 counter = 8
Thread 2 counter = 5
Thread 1 counter = 9
Thread 1 counter = 10…
```

Listing 2 A first pthread example (Example: pthread_static_example_1)

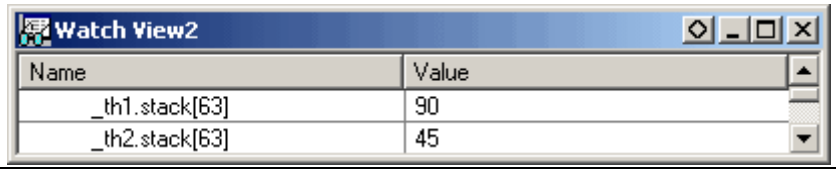


Figure 1 Snapshot of the debugger watch view

4.2 Creating Threads

Each thread is associated with a thread-control block. The header file pthread.h defines the following type

```
1  typedef struct pthread_s {
2      struct pthread_s *next; //!< next thread pointer
3      struct pthread_s *prev; //!< previous thread pointer
4      uint32_t lcx;           //!< lower context pointer
5      uint32_t priority;      //!< thread priority. 0 to PTHREAD_PRIO_MAX
6      uint32_t policy;        //!< policy is one of sched_policy_e
7      uint32_t *arg;          //!< argument passed at thread start
8      uint32_t stack[1];      //!< stack[1] is only a dummy
   }*pthread_t;
```

Listing 3 Thread-control block pthread_t type definition.

The blocks are organized as a double linked list. The block holds a pointer to the current lower context (Line 3), the threads priority (Line 4) and policy (Line 5). The priority is number ranged from 0 to PTHREAD_PRIO_MAX. When multiple threads are runnable, the scheduler determines the thread with the highest priority. The library supports first-in-first-out (SCHED_FIFO) and round-robin (SCHED_RR) policy. The last element of the pthread_s structure is the stack. The type definition only defines a dummy value. The actual definition of a thread-control block uses more memory.

All threads are defined as static, i.e. for each thread function there must be a thread-control block defined. The macro PTHREAD_CONTROL_BLOCK is used to define a TCB. It defines the name, priority, policy, and stack size.

Example:

```
PTHREAD_CONTROL_BLOCK(th1,2,SCHED_FIFO,PTHREAD_DEFAULT_STACK_SIZE)
```

The thread-control block named th1 is defined with thread priority 2, SCHED_FIFO policy, and a default stack size.

This thread-control block is passed to the pthread_create_np() function, where the POSIX version dynamically creates the structure, i.e. allocates memory on the heap. The pthread_create_np() function reserves two context areas (Listing 4, Line 11-13), from a block allocated in cstart for the thread and sets up the PSW, the stack pointer register A10 and Program Counter PC. Finally it puts the threads control block in a linked list. The library organizes the thread-control blocks in an array of linked lists.

```
pthread_t pthread_runnable_threads[PTHREAD_PRIO_MAX];
```

If the array element is not empty but holds one thread-control block a corresponding bit in the 32-bit pthread_runnable variable is set. Therefore the scheduler can quickly find the thread with the highest priority using the CLZ instruction.

```
 1 int pthread_create_np(pthread_t thread, const pthread_attr_t *attr,
 2                       void(*start_routine)(void *), void *arg)
 3 {
 4     const pthread_attr_t default_attr= PTHREAD_DEFAULT_ATTR;
 5     uint32_t fcx;
 6     context_t *cx;
 7
 8     if (attr == NULL)
 9         attr = &default_attr;
10
11     fcx = __mfcr(FCX);           // At start-up the context is a linear array
12     thread->lcx = fcx - 1;       // so that a decrement of 2 reserves an
13     __mtcr(FCX, fcx - 2);          // upper and lower context.
14
15     cx = cx_to_addr(fcx);        // Convert context pointer to address
16     cx->u.psw  = 0 << 12         // Protection Register Set PRS=0
17                | attr->mode << 10                // I/O Privilege
18                | 1L << 7         // Call depth counting is enabled CDE=1
19                | attr->call_depth_overflow;      // Call Depth Overflow
20     cx->u.a10  = thread->stack + *thread->stack; // stack grow down
21     cx->u.a11  = 0;              // New task has no return address
22     cx->u.pcxi = 0;              // No previous context
23     cx--;                        // Decrement to get the lower context address
24     cx->l.pcxi = 0L << 24        // Previous CPU Priority Number PCPN=0
25                | 1L << 23        // Previous Interrupt Enable PIE=1
26                | 1L << 22        // Upper Context Tag.
27                | fcx;            // Previous Context Pointer
28     cx->l.pc   = start_routine; // Init new task start address
29     cx->l.a4   = arg;            // Argument when thread started
30     thread->arg = arg;           // Container that saves the argument
31
32     uint32_t i = thread->priority;
33     list_append(&pthread_runnable_threads[i], thread, thread,
34                 pthread_runnable_threads[i]);
35     __putbit(1,(int*)&pthread_runnable,i); // mark current thread as runnable
36     return 0;
}
```

Listing 4 pthread_create_np function.

The context save area set-up in the LDRAM after the first 2 threads were created is shown in Figure 2. The thread occupies a lower and upper context. The lower context PCXI is linked to the upper context. The contexts program counter variable PC is initialized with the thread start address. The contexts A4 variable holds the argument which is passed to the thread program when it is called the first time. The upper context block holds the pointer to the upper limit of the stack. On the TriCore the stack grows downward. The contexts A11, the return address is initialized to NULL, because the threads in this library run forever, i.e. never return.

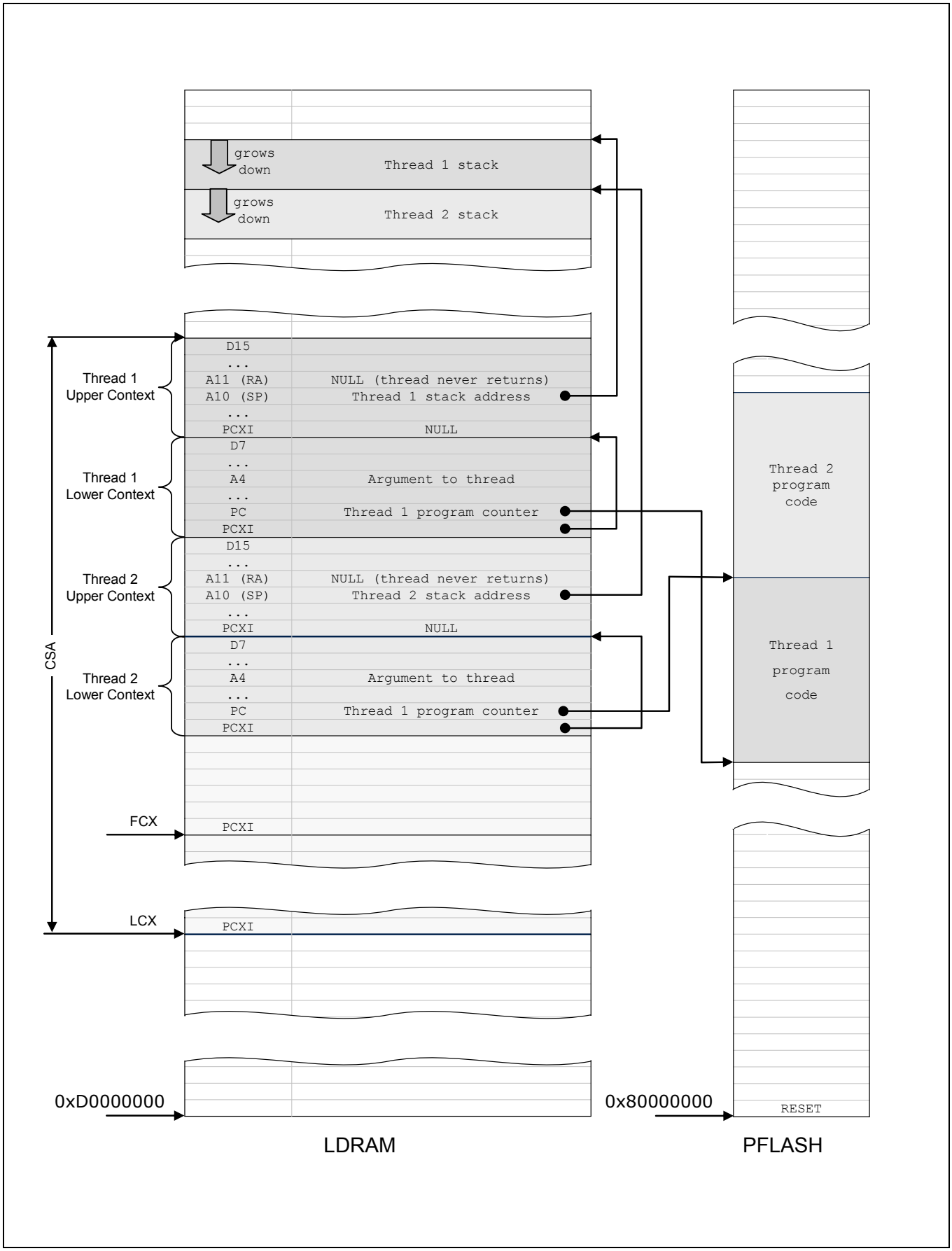


Figure 2 Context set-up after first 2 threads were created.

4.3 Starting Threads

The function pthread_create_np() does not start the thread. This is done with the function pthread_start_np(). In Listing 4, the threads are started after the system timer initialization. An initialization of the system timer by pthread_schedrr_init_np() is required because of the round-robin scheduling policy in the example. The function pthread_create_np() sets the previous context (Line 22) to the thread with the highest priority (Listing 5 Line 10) before restoring the lower context (Line 23) and the upper context by calling Return From Event (RFE). RFE also enables the interrupt system, i.e. the system-timer interrupt used by the round-robin scheduler.

The service-request node is only enabled when there is more than 1 thread at the current running priority and the current thread has a round-robin scheduling policy (Lines 14-17). This avoids the overhead of interrupting the current thread by the system timer when there is no other thread. The list-handling functions (see list_append, list_delete_first in pthread.c) take care of setting the next member to NULL is there is only one list element.

```
 1 inline void pthread_start_np(void)
 2 {
 3     extern uint32_t pthread_runnable;
 4     extern pthread_t pthread_running;
 5     extern pthread_t, pthread_runnable_threads[PTHREAD_PRIO_MAX];
 6
 7     pthread_t thread;
 8
 9     // get ready thread with highest priority ready
10     thread = pthread_runnable_threads[31 - clz(pthread_runnable)];
11
12     // check if timer must be enabled if thread policy is SCHED_RR
13     // and there is another thread with the same priority
14     if (thread->next != NULL && thread->policy == SCHED_RR)
15         STM_SRC0.B.SRE = 1;
16     else
17         STM_SRC0.B.SRE = 0;
18
19     PTHREAD_SWAP_HANDLER(thread,pthread_running); // callback hook (optional)
20
21     pthread_running = thread;
22     __dsync();                  // Required before mtcr
23     __mtcr(PCXI,thread->lcx);   // Set previous context to start thread
24     __rslcx();                  // Restore the lower context
25     __asm(" mov d2,#0");
26     __asm(" rfe");              // Return and restore upper context and enable
27 }
```

Listing 5 pthread_start_np function

4.4 Scheduling Threads

The pthread timer in the example is setup to generate an interrupt every increment of STM_TIM4. With the reset value the STM module runs at ¼ of the CPU frequency, i.e. 45 MHz when the TC1797 [3] runs at a maximum

speed of 180 MHz. The interrupt is generated every 728 µs. The interrupt uses the service-request node 0 with priority SCHEDRR_INT (Line 6).

```
 1 inline void pthread_schedrr_init_np(void)
 2 {
 3     STM_CMP0.U = 1;            // load compare register 0 with constant 1
 4     STM_CMCON.B.MSIZE0 = 0;    // CMP0[0] used for compare operation
 5     STM_CMCON.B.MSTART0 = 16; // STM[16] is the lowest bit number (STM_TIM4)
 6                               // 2^(16-1)/45MHz = 728us
 7     STM_ICR.B.CMP0EN = 1;      // Interrupt on compare match with CMP0 enabled
 8     STM_ICR.B.CMP0OS = 0;      // Interrupt output STMIR0 selected
 9     STM_SRC0.U = 0x1000 | SCHEDRR_INT; // set and enable service request
10 }
```

Listing 6 pthread_schedrr_init_np function

The implementation of the interrupt handler is shown in Listing 7. The interrupt handler is directly placed in the interrupt vector table with the __interrupt_fast() qualifier. Every time the handler is called the next thread in the linked list is made the running thread (Line 5). If this thread does not have a round-robin scheduling policy the system timer service-request node will be disabled (Lines 8-11).

```
 1 void stm_src0 (void) {
 2     pthread_t thread;
 3
 4     pthread_running->lcx = __mfcr(PCXI);
 5     thread = pthread_running->next;
 6     pthread_runnable_threads[thread->priority] = thread;
 7
 8     if (thread->policy == SCHED_RR)
 9         STM_SRC0.B.SRE = 1;
10     else
11         STM_SRC0.B.SRE = 0;
12
13     PTHREAD_SWAP_HANDLER(thread,pthread_running); // callback hook (optional)
14
15     pthread_running = thread;
16     __dsync(); // required before PCXI manipulation
17     __mtcr(PCXI,thread->lcx);     // modify previous context
18     __asm("ji a11");
19 }
20
21 void __interrupt_fast(2) stm_src0_fast(void) { // store upper context
22     STM_ISRR.B.CMP0IRR = 1;       // enable interrupts again
23     __asm(" svlcx          \n"    // store lower context
24           " jla stm_src0 \n"     // jump and link absolute to stm_src0
25           " rslcx           ");  // restore lower context
26                                  // RFE restores upper context
   }
```

Listing 7 stm_src0 interrupt function

4.5 Dispatching Threads

The thread dispatcher is invoked by a system call SYSCALL which causes a system trap (Listing 8). The trap identification number TIN defines two cases: DISPATCH_WAIT (Line 38) or DISPATCH_SIGNAL (Line 43). Either the current running thread is swapped out or one or more blocked threads are appended to the runnable thread list. In both cases the function pthread_start_np() identifies the thread with the highest priority number in the runnable thread list (Line 9) and swaps in this thread.

```
//! Start threads
inline void pthread_start_np(void) {
    extern uint32_t pthread_runnable;
    extern pthread_t pthread_running;
    extern pthread_t, pthread_runnable_threads[PTHREAD_PRIO_MAX];
    pthread_t thread;

    //  get ready thread with highest priority ready
    thread = pthread_runnable_threads[31-clz(pthread_runnable)];

    // check if timer must be enabled if thread policy is SCHED_RR
    // and there is another thread with the same priority
    if (thread->next != NULL && thread->policy == SCHED_RR)
        STM_SRC0.B.SRE = 1; // STOREBIT(STM_SRC0, 12, 1);
    else
        STM_SRC0.B.SRE = 0; // STOREBIT(STM_SRC0, 12, 0);

    pthread_running = thread;
    __dsync();
    __mtcr(PCXI, thread->lcx);
    __rslcx();
    __asm(" mov d2,#0");
    __asm(" rfe");
}

static void trapsystem(pthread_t *blocked_threads_ptr, pthread_t last_thread)
{
    int tin, i;
    pthread_t thread, tmp;

    __asm(" mov %0,d15 \n"
          " svlcx       "
          :"=d"(tin)); // put d15 in C variable tin

    pthread_running->lcx = __mfcr(PCXI);
    i = pthread_running->priority;
    switch (tin) {
    case DISPATCH_WAIT: // _swap_out _pthread_running
      list_delete_first(&pthread_runnable_threads[i]);
      list_append(blocked_threads_ptr,pthread_running,pthread_running,NULL);
      __putbit(neza(pthread_runnable_threads[i]),(int*)&pthread_runnable,i);
        break;
    case DISPATCH_SIGNAL: // append blocked threads
      tmp = NULL;
      assert(blocked_threads_ptr);
      thread = *blocked_threads_ptr;
      while (thread != NULL) {
          tmp = thread->next;
          i = thread->priority;
          list_append(&pthread_runnable_threads[i], thread, thread,
                  pthread_runnable_threads[i]);
          __putbit(1,(int*)&pthread_runnable,i);
          if (thread == last_thread)
              break;
          thread = tmp;
      }
      *blocked_threads_ptr = tmp;
      break;
    default:
      break;
    }
    pthread_start_np();
}
```

Listing 8 thread dispatcher function trapsystem()

4.6 Condition scheduling

The pthread library implements the pthread_cond_timedwait_np() function which blocks the thread until the condition is signaled or until a timeout period elapsed. To use this function, a timer must be initialized by calling pthread_cond_timedwait_init_np() (Listing 9). The pthread_cond_timedwait_np() is a non-portable POSIX implementation using a relative time parameter. The relative time has to be given in 1 to 65536 ticks of the system timer of 728 µs.

```
1 inline void pthread_cond_timedwait_init_np() {
2     STM_CMCON.B.MSIZE1 = 15;  // 16 bits are used for compare operation
3     STM_CMCON.B.MSTART1 = 16; // STM[16:31] that is compared to CMP1
4                               // Interrupt period 1 to 65535 ticks à 728us
5     STM_ICR.B.CMP1EN = 1;     // Interrupt on compare match with CMP1 enabled
6     STM_ICR.B.CMP1OS = 1;     // Interrupt output STMIR1 selected
7     STM_SRC1.U = TIMEDWAIT_INT ; // set service request control
8 }
```

Listing 9 pthread_schedrr_init_np function

5 API Reference

The library implements a subset of the POSIX pthread standard [4]. Non-portable functions have a _np suffix. These functions might have different parameter or different behaviour, or they are not available in the standard, In general the implementation does not set the error number. For a complete description of the pthread functions please read the Open Group Base Specifications Issue 6, IEEE Std 1003.1 (see www.opengroup.org).

Table 1 Compatibility Matrix

Name	Notes
pthread_create_np (pthread_t thread, const pthread_attr_t *attr, void(*start_routine)(void *), void *arg)	Non-portable version of pthread_create. - Function must be called from main. - The function does not start the thread. - The parameter thread does not hold the ID upon successful completion, but is the thread-control block that must be allocated in advance. - The thread startroutine never returns.
pthread_mutex_trylock(pthread_mutex_t *mutex)	-
pthread_mutex_unlock(pthread_mutex_t *mutex)	-
pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex)	-
pthread_cond_timedwait_np(pthread_cond_t *cond, pthread_mutex_t *mutex, uint16_t reltime)	Non-portable version of pthread_timewait - The time to wait is specified by the reltime parameter as a relative system time. The maximum number of conditions that can wait is defined by TW_SZ.
pthread_cond_broadcast(pthread_cond_t *cond)	-
pthread_cond_signal(pthread_cond_t *cond)	-
pthread_cond_timedwait_init_np(void)	Non-portable function that is not available in the POSIX standard. The timer is required in case of - timed wait on condition. The initialization of the timer should be done in after thread creation.
pthread_schedrr_init_np(void)	Non-portable function that is not available in the POSIX standard. The timer is required in case of - round robin scheduling policy. The initialization of the timer should be done after thread creation.
pthread_start_np(void)	Non-portable function that is not available in the POSIX standard. This function should be called at the end of main and starts the thread execution. It never returns.

6 Synchronizing

The library implements mutex and conditions variables. To protect a shared resource from a race condition, a type of synchronization called mutual exclusion, or mutex for short can be used. Using mutexes a thread turns at having exclusive access to data. When one thread has exclusive access to data, other threads cannot simultaneously access the same data. The mutex is similar to the principle of the binary semaphore with one significant difference: the principle of ownership. Ownership is the simple concept that when a task locks (acquires) a mutex, only the same task can unlock (release) it. If a task tries to unlock a mutex it hasn't locked (thus doesn't own) then an error condition is encountered and, most importantly, the mutex is not unlocked. If the mutual exclusion object doesn't have ownership then, it is not a mutex no matter what it is called.

The concept of ownership enables mutex implementations to address the problems and inherent dangers associated with using the semaphores: accidental release, recursive deadlock, task-death deadlock, priority inversion and using a semaphore as a signal. Semaphores are rarely required in embedded systems software.

Whereas a mutex allows threads to synchronize by controlling their access to data, a condition variable allows threads to synchronize on the value of data. The POSIX condition implements a unilateral synchronization. A task waits for another task until is signals an event. Bilateral synchronization, often called a rendezvous is rarely supported by RTOSs. A condition variable provides a way of naming an event in which threads have a general interest. An event can be as simple as a counter's reaching a particular value or a flag being set or cleared. Pthreads conditions are a perfect choice when a thread waits for a resource, i.e. an interrupt to provide new data.

6.1 Mutex variables Example: pthread_static_example_2

This example shows the corruption that can result if no serialization is done and also shows the use of pthread_mutex_lock(). It can be called with no parameters to use pthread_mutex_lock() to protect the critical section, or called with one or more parameters to show data corruption that occurs without locking.

The example creates four threads with round-robin scheduling policy with the same thread function. The threads increment four global variables i, j, k, l. Without using a mutex, there is a chance that a thread switch will happen after loading a global variable. The new thread increments the variable and when the first thread continues executing the value that is already loaded to a register is outdated. This results in different values of i, j, k, and l, as shown in the first output window.

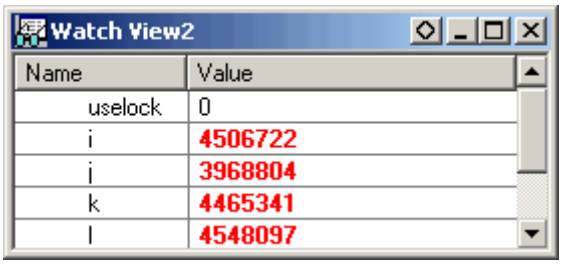
The second output window shows the values when the critical section of incrementing the global variables is protected using a mutex.

Example 2

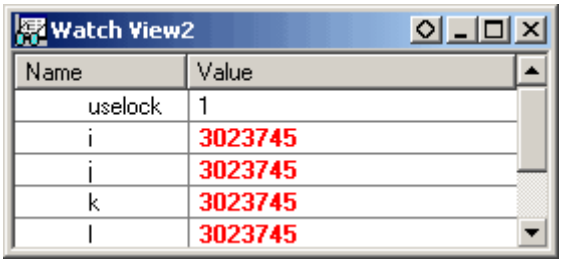
```
 1 #pragma align 8
 2 // define thread name, priority, policy, stack size
 3 PTHREAD_CONTROL_BLOCK(th1,1,SCHED_RR,PTHREAD_DEFAULT_STACK_SIZE)
 4 PTHREAD_CONTROL_BLOCK(th2,1,SCHED_RR,PTHREAD_DEFAULT_STACK_SIZE)
 5 PTHREAD_CONTROL_BLOCK(th3,1,SCHED_RR,PTHREAD_DEFAULT_STACK_SIZE)
 6 PTHREAD_CONTROL_BLOCK(th4,1,SCHED_RR,PTHREAD_DEFAULT_STACK_SIZE)
 7 #pragma align restore
 8
 9 pthread_mutex_t    mutex = PTHREAD_MUTEX_INITIALIZER;
10 int                i,j,k,l;
11 int volatile       uselock=0;
12
13 void thread(void* arg) {
14     for (;;) {
15         if (uselock)
16             pthread_mutex_lock(&mutex);
17         ++i; ++j; ++k; ++l;
18         if (uselock)
```

```
            pthread_mutex_unlock(&mutex);
    }
}

void main(void) {
    pll_init();

    printf("Create 4 threads with round-robin policy.\n");

    pthread_create_np(th1, NULL, thread, (void*) 1);
    pthread_create_np(th2, NULL, thread, (void*) 2);
    pthread_create_np(th3, NULL, thread, (void*) 2);
    pthread_create_np(th4, NULL, thread, (void*) 2);

    pthread_init_timer_np();
    pthread_start_np();
}
```

Output without using mutex (Variable unlock=0)



Output with using mutex (Variable unlock=1)



Listing 10 Synchronizing example using mutex

6.2 Conditions variables Example: pthread_static_example_3

This example shows how to use a condition variable to wake up a thread. All seven threads are created with priority 2, first-in-first-out scheduling policy and the same thread function thread2. These threads are waiting on the condition variable conditionMet which is initialized with 0 to become 1. So all threads become blocked by the condition until thread th0 with priority 1 and thread function thread1 are running and conditionMet is set to 1. A call of pthread_cond_broadcast signals all blocked threads to become runnable.

Example 3

```
// define thread name, priority, policy, stack size
PTHREAD_CONTROL_BLOCK(th0,1,SCHED_FIFO,PTHREAD_DEFAULT_STACK_SIZE)
PTHREAD_CONTROL_BLOCK(th1,2,SCHED_FIFO,PTHREAD_DEFAULT_STACK_SIZE)
PTHREAD_CONTROL_BLOCK(th2,2,SCHED_FIFO,PTHREAD_DEFAULT_STACK_SIZE)
```

```
PTHREAD_CONTROL_BLOCK(th3,2,SCHED_FIFO,PTHREAD_DEFAULT_STACK_SIZE)
PTHREAD_CONTROL_BLOCK(th4,2,SCHED_FIFO,PTHREAD_DEFAULT_STACK_SIZE)
PTHREAD_CONTROL_BLOCK(th5,2,SCHED_FIFO,PTHREAD_DEFAULT_STACK_SIZE)
PTHREAD_CONTROL_BLOCK(th6,2,SCHED_FIFO,PTHREAD_DEFAULT_STACK_SIZE)
PTHREAD_CONTROL_BLOCK(th7,2,SCHED_FIFO,PTHREAD_DEFAULT_STACK_SIZE)

int32_t volatile conditionMet = 0;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void thread1(void* arg) {
    for (;;) {
        pthread_mutex_lock(&mutex);
        conditionMet = 1;
        puts("Wake up all waiters...");
        delay_ms(200);
        pthread_cond_broadcast(&cond);
        pthread_mutex_unlock(&mutex);
    }
}

void thread2(void* arg) {
    for (;;) {
        pthread_mutex_lock(&mutex);
        conditionMet = 0;
        while (!conditionMet) {
            printf("Thread %d blocked\n", (int) arg);
            delay_ms(200);
            pthread_cond_wait(&cond, &mutex);
        }
        pthread_mutex_unlock(&mutex);
    }
}

void main(void) {
    pll_init();

    pthread_create_np(th0, NULL, thread1, (void*) 0);
    pthread_create_np(th1, NULL, thread2, (void*) 1);
    pthread_create_np(th2, NULL, thread2, (void*) 2);
    pthread_create_np(th3, NULL, thread2, (void*) 3);
    pthread_create_np(th4, NULL, thread2, (void*) 4);
    pthread_create_np(th5, NULL, thread2, (void*) 5);
    pthread_create_np(th6, NULL, thread2, (void*) 6);
    pthread_create_np(th7, NULL, thread2, (void*) 7);

    pthread_start_np();
}
```

Output

```
Thread 1 blocked
Thread 2 blocked
Thread 3 blocked
Thread 4 blocked
Thread 5 blocked
Thread 6 blocked
Thread 7 blocked
Wake up all waiters
...
```

Listing 11 Synchronizing example using conditions

6.3 Condition variables from interrupt handler Example: pthread_static_example_4

This example shows how to use a condition variable to wake up a thread from an interrupt. Thread th1 and th2 are waiting for a condition (Line 23) and become blocked. The receive interrupt handler from an ASC interface broadcasts the condition (Line 14) and make the threads runnable again.The example also implements a thread on the lowest priority which is executed on idle.

Example 4

```
 1 // define thread name, priority, policy, stack size
 2 PTHREAD_CONTROL_BLOCK(th0,0,SCHED_FIFO,PTHREAD_DEFAULT_STACK_SIZE)
 3 PTHREAD_CONTROL_BLOCK(th1,2,SCHED_FIFO,PTHREAD_DEFAULT_STACK_SIZE)
 4 PTHREAD_CONTROL_BLOCK(th2,2,SCHED_FIFO,PTHREAD_DEFAULT_STACK_SIZE)
 5
 6 pthread_cond_t cond   = PTHREAD_COND_INITIALIZER;
 7 pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
 8
 9 void __interrupt(5) asc0_rx(void) {
10     puts("Wake up all waiters...");
11     pthread_cond_broadcast(&cond);
12 }
13
14 void idle(void* arg) {
15     for (;;)
16         ;
17 }
18
19 void thread(void* arg) {
20     for (;;) {
21         pthread_mutex_lock(&mutex);
22         printf("Thread %d blocked\n", (int) arg);
23         pthread_cond_wait(&cond, &mutex);
24         printf("Thread %d continued\n", (int) arg);
25         pthread_mutex_unlock(&mutex);
26     }
27 }
28
29 void main(void) {
30     pll_init();
31     asc0_init();
32
33     pthread_create_np(th0, NULL, idle, (void*) 0);
34     pthread_create_np(th1, NULL, thread, (void*) 1);
35     pthread_create_np(th2, NULL, thread, (void*) 2);
36
37     pthread_start_np();
38 }
```

Output

```
Thread 1 blocked
Thread 2 blocked
Wake up all waiters...
Thread 1 continued
Thread 1 blocked
Thread 2 continued
Thread 2 blocked
Wake up all waiters...
```

Listing 12 Synchronizing example using condition with a wake-up from an interrupt

6.4 Condition variables with timeout period Example: pthread_static_example_5

This example shows how to use a condition variable to wake up a thread from an interrupt or after a timeout period has elapsed. Thread th1 and th2 are waiting for a condition (Lines 26, 35) and become blocked. The receive interrupt handler from an ASC interface broadcasts the conditions (Lines 15-16) and make the threads runnable again. Thread th2 is blocked with a timeout period of 100 timer ticks (Line 35).

Example 5

```
1  // define thread name, priority, policy, stack size
2  PTHREAD_CONTROL_BLOCK(th0,0,SCHED_FIFO,PTHREAD_DEFAULT_STACK_SIZE)
3  PTHREAD_CONTROL_BLOCK(th1,2,SCHED_FIFO,PTHREAD_DEFAULT_STACK_SIZE)
4  PTHREAD_CONTROL_BLOCK(th2,2,SCHED_FIFO,PTHREAD_DEFAULT_STACK_SIZE)
5
6  pthread_cond_t cond1 = PTHREAD_COND_INITIALIZER;
7  pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
8  pthread_cond_t cond2 = PTHREAD_COND_INITIALIZER;
9  pthread_mutex_t mutex2 = PTHREAD_MUTEX_INITIALIZER;
10
11 void __interrupt(5) asc0_rx(void) {
12     puts("Wake up all waiters...");
13     pthread_cond_broadcast(&cond1);
14     pthread_cond_broadcast(&cond2);
15 }
16
17 void idle(void* arg) {
18     for (;;)
19         ;
20 }
21
22 void thread1(void* arg) {
23     for (;;) {
24         pthread_mutex_lock(&mutex1);
25         printf("Thread %d blocked\n", (int) arg);
26         pthread_cond_wait(&cond1, &mutex1);
27         printf("Thread %d continued\n", (int) arg);
28         pthread_mutex_unlock(&mutex1);
29     }
30 }
31 void thread2(void* arg) {
32     for (;;) {
33         pthread_mutex_lock(&mutex2);
34         printf("Thread %d blocked\n", (int) arg);
35         pthread_cond_timedwait_np(&cond2, &mutex2, 100);
36         printf("Thread %d continued\n", (int) arg);
37         pthread_mutex_unlock(&mutex2);
38     }
39 }
40
41 void main(void) {
42     pll_init();
43     asc0_init();
44
45     printf("Example 5: Create 3 threads with first-in-first-out policy."
46            "Shows how to block a thread until the condition is signaled or"
47            " until a timeout period elapsed.\n");
48
49     pthread_create_np(th0, NULL, idle, (void*) 0);
50     pthread_create_np(th1, NULL, thread1, (void*) 1);
51     pthread_create_np(th2, NULL, thread2, (void*) 2);
52
53     pthread_cond_timedwait_init_np(); // timedwait condition requires a timer
54     pthread_start_np();
```

```
55  }
```

Output

```
Thread 1 blocked
Thread 2 blocked
Thread 2 continued
Thread 2 blocked
Wake up all waiters...
Thread 1 continued
Thread 1 blocked
Thread 2 continued
Thread 2 blocked
Thread 2 continued
```

Listing 13 Synchronizing example using condition with a timeout period

7 Tools

The pthread library is built using Tasking 3.3r1. The example code includes project workspaces for the PLS UDE debugger V2.6.11.

8 Source code

The source code provided with this application consists of a library project pthread_static and five example programs that were generally described in the chapter 4 and 6. Documentation is extracted directly from the sources using eclox [5]. Open the \html\index.html in each project directory.

9 References

[1] TriCore Architecture V1.3.8 2007-11

[2] http://www.infineon.com/tricore

[3] TC1797 User's Manual V1.1 2009-05

[4] The Open Group Base Specifications Issue 6, IEEE Std 1003.1, http://www.opengroup.org

[5] Eclox, a Doxygen frontend plugin for Ecplise. http://home.gna.org/eclox and http://www.doxygen.org

w w w . i n f i n e o n . c o m

Published by Infineon Technologies AG